Creating and Destroying Objects

ITEM 1 : CONSIDER STATIC FACTORY METHODS INSTEAD OF CONSTRUCTORS

All of objects creation were through factory methods such as:

- Employee.of() , for creating a 'Employee'
- <U> Cacheable.of() , for creating a Cacheable<U>'

ITEM 2: CONSIDER A BUILDER WHEN FACED WITH MANY CONSTRUCTOR PARAMETERS

I didn't have to use such types because most of the classes had either 1 or 2 constructors, and to keep fields immutable, so the data are thread safe.

ITEM 3: ENFORCE THE SINGLETON PROPERTY WITH A PRIVATE CONSTRUCTOR OR AN ENUM TYPE

I enforce the singleton property with a private constructor such as:

In the Client:

- AppDataManager(), a single AppDataManager instance is needed for the whole client application.
- AppDBHelper(), a single AppDBHelper instance is needed for the whole client application.
-

In the Server:

- EmployeeCache, a single EmployeeCache instance is needed for the whole client application.
- BasicConnectionPool, a single EmployeeCache instance is needed for the whole client application.

ITEM 4: ENFORCE NONINSTANTIABILITY WITH A PRIVATE CONSTRUCTOR

For all the utility classes are final and have their default constructor private to enforce non-instantiability.

I also used this idea to enforce using the factory methods instead of calling constructors.

ITEM 5: PREFER DEPENDENCY INJECTION TO HARDWIRING RESOURCES

In the client:

- Gson instance injects to AppDBHelper
- DBHelper instance injects to AppDataManager
- DataManager instance injects to AppMainController
- MainController instance injects to MainView

In the Server:

- EmployeeCache instance injects to ClientRunnable
- BasicConnectionPool instance injects to ClientRunnable

ITEM 6: AVOID CREATING UNNECESSARY OBJECTS

I tried to create as little objects as possible.

ITEM 7: ELIMINATE OBSOLETE OBJECT REFERENCES

I mostly used `List` which is supposed to handle the memory issues.

ITEM 8: AVOID FINALIZERS AND CLEANERS

My code didn't use any of them.

ITEM 9: PREFER TRY-WITH-RESOURCES TO TRY-FINALLY

All the `try-catch` statements that I used with `Closable` objects were `try-with-resources` such as:

- `Socket` used to establish a connection on the client side.
- `SocketReader` used to read values from socket.
- `SocketWriter` used to write values to socket.

Methods Common to All Objects

ITEM 10: OBEY THE GENERAL CONTRACT WHEN OVERRIDING EQUALS

All the `equals()` methods were implemented through `intellij` auto-generated code.

ITEM 11: ALWAYS OVERRIDE HASHCODE WHEN YOU OVERRIDE EQUALS

All the classes were `equals()` were overridden, `hashCode()` was overridden as well.

ITEM 12: ALWAYS OVERRIDE TOSTRING

All the instantiable classes have a `toString()` method.

ITEM 13: OVERRIDE CLONE JUDICIOUSLY

There was no need for the `clone()` method.

ITEM 14: CONSIDER IMPLEMENTING COMPARABLE

I implement `Comparable` on:

- Cacheable: to compare with another Cacheable based on the cache priority

Classes and Interfaces

ITEM 15: MINIMIZE THE ACCESSIBILITY OF CLASSES AND MEMBERS

The access for all the classes and fields were set to minimum.

ITEM 16: IN PUBLIC CLASSES, USE ACCESSOR METHODS, NOT PUBLIC FIELDS

all non-final fields were accessed through `getter/setters` in all the classes.

ITEM 17: MINIMIZE MUTABILITY

I tried to keep my classes and fields immutable, such as:

- Employee
- AppDataManager and AppDBHelper
- Request, ByIdRequest, EntityByIdRequest and EntityRequest
- Cacheable
- EmployeeCache
- BasicConnectionPool

ITEM 18: FAVOR COMPOSITION OVER INHERITANCE

Not applicable to the project.

ITEM 19: DESIGN AND DOCUMENT FOR INHERITANCE OR ELSE PROHIBIT IT

In my code, you can inherit:

- `Request`, so that you can create a new type of request.
- `Query`, so that you can create a new type of queries.
- `Repository`, so that you can create a new type of repositories.
- `ICache`, so that you can create a new type of caches.
- `ConnectionPool`, so that you can create a new type of connectionpools.

ITEM 20: PREFER INTERFACES TO ABSTRACT CLASSES

I created Query, Repository, ICache, ConnectionPool to upper-limit the generic data types.

ITEM 21: DESIGN INTERFACES FOR POSTERITY

All of the interfaces are fully designed for posterity.

ITEM 22: USE INTERFACES ONLY TO DEFINE TYPES

All of the interfaces define a type only.

ITEM 23: PREFER CLASS HIERARCHIES TO TAGGED CLASSES

Not applicable to the project.

ITEM 24: FAVOR STATIC MEMBER CLASSES OVER NONSTATIC

There are no inner-classes in my code.

ITEM 25: LIMIT SOURCE FILES TO A SINGLE TOP-LEVEL CLASS

I have only one top-level class in all the files.

Generics

ITEM 26: DON’T USE RAW TYPES

In all my uses `Cacheable`, `ICache`, `SocketReader`, `SocketWriter`, `Repository` and `Query` I defined them as generic and in the subclass I specified exactly what the type is.

ITEM 27: ELIMINATE UNCHECKED WARNINGS

All of the warnings are put in account.

ITEM 28: PREFER LISTS TO ARRAYS

There are no arrays in my code.

ITEM 29: FAVOR GENERIC TYPES

I used generic types in `Cacheable`, `ICache`, `SocketReader`, `SocketWriter`, `Repository` and `Query`.

ITEM 30: FAVOR GENERIC METHODS

I didn't need that in my project.

ITEM 31: USE BOUNDED WILDCARDS TO INCREASE API FLEXIBILITY

I didn't need that in my project.

ITEM 32: COMBINE GENERICS AND VARARGS JUDICIOUSLY

Not applicable to the project.

ITEM 33: CONSIDER TYPESAFE HETEROGENEOUS CONTAINERS

Not applicable to the project.

Enums and Annotations

ITEM 34: USE ENUMS INSTEAD OF INT CONSTANTS

I used RequestType to represent the type of the request instead of constants.

ITEM 35: USE INSTANCE FIELDS INSTEAD OF ORDINALS

Not applicable to the project.

ITEM 36: USE ENUMSET INSTEAD OF BIT FIELDS

Not applicable to the project.

ITEM 37: USE ENUMMAP INSTEAD OF ORDINAL INDEXING

Not applicable to the project.

ITEM 38: EMULATE EXTENSIBLE ENUMS WITH INTERFACES

Not applicable to the project.

ITEM 39: PREFER ANNOTATIONS TO NAMING PATTERNS

Not applicable to the project.

ITEM 40: CONSISTENTLY USE THE OVERRIDE ANNOTATION

All Overridden methods were tagged with the `Override` annotation.

ITEM 41: USE MARKER INTERFACES TO DEFINE TYPES

Not applicable to the project.

Lambdas and Streams

ITEM 42: PREFER LAMBDAS TO ANONYMOUS CLASSES

I used lambdas when it was applicable to use.

ITEM 43: PREFER METHOD REFERENCES TO LAMBDAS

I used method references in most of my cases.

ITEM 44: FAVOR THE USE OF STANDARD FUNCTIONAL INTERFACES

I used `Function` interface to represent a function in my code.

ITEM 45: USE STREAMS JUDICIOUSLY

Not applicable to the project.

ITEM 46: PREFER SIDE-EFFECT-FREE FUNCTIONS IN STREAMS

Not applicable to the project.

ITEM 47: PREFER COLLECTION TO STREAM AS A RETURN TYPE

Not applicable to the project.

ITEM 48: USE CAUTION WHEN MAKING STREAMS PARALLEL

Not applicable to the project.

Methods

ITEM 49: CHECK PARAMETERS FOR VALIDITY

The parameters are always validated before taken into account.

ITEM 50: MAKE DEFENSIVE COPIES WHEN NEEDED

The parameters are always validated before taken into account.

ITEM 51: DESIGN METHOD SIGNATURES CAREFULLY

The signatures are clear to the readers.

ITEM 52: USE OVERLOADING JUDICIOUSLY

Overloaded functions used in `Employee` factory classes were well defined.

ITEM 53: USE VARARGS JUDICIOUSLY

Used in insertion multiple employees to the database.

ITEM 54: RETURN EMPTY COLLECTIONS OR ARRAYS, NOT NULLS

I used empty optional instead of nulls.

ITEM 55: RETURN OPTIONALS JUDICIOUSLY

I used optional when returning an employee from the database by id.

ITEM 56: WRITE DOC COMMENTS FOR ALL EXPOSED API ELEMENTS

Doc comments were written for all exposed and non-exposed elements.

General Programming

ITEM 57: MINIMIZE THE SCOPE OF LOCAL VARIABLES

I tried doing that by defining local variables in the smallest scope.

ITEM 58: PREFER FOR-EACH LOOPS TO TRADITIONAL FOR LOOPS

I always used a for each since I didn't need the traditional one.

ITEM 59: KNOW AND USE THE LIBRARIES

Used the available libraries to convert between objects and json.

ITEM 60: AVOID FLOAT AND DOUBLE IF EXACT ANSWERS ARE REQUIRED

Not applicable to the project.

ITEM 61: PREFER PRIMITIVE TYPES TO BOXED PRIMITIVES

I always used primitive types since I didn't need the boxed ones.

ITEM 62: AVOID STRINGS WHERE OTHER TYPES ARE MORE APPROPRIATE

Not applicable to the project.

ITEM 63: BEWARE THE PERFORMANCE OF STRING CONCATENATION

Not applicable to the project.

ITEM 64: REFER TO OBJECTS BY THEIR INTERFACES

I always used interfaces to refer to objects.

ITEM 65: PREFER INTERFACES TO REFLECTION

Not applicable to the project.

ITEM 66: USE NATIVE METHODS JUDICIOUSLY

Not applicable to the project.

ITEM 67: OPTIMIZE JUDICIOUSLY

My project is open for posterity.

ITEM 68: ADHERE TO GENERALLY ACCEPTED NAMING CONVENTIONS

I always followed the naming conventions while naming classes, interfaces, fields, variables, methods, packages …

Exceptions

ITEM 69: USE EXCEPTIONS ONLY FOR EXCEPTIONAL CONDITIONS

I throw a `Runtime Exception` when something wrong happens, I follow the `Do Not Swallow The Exceptions` rule.

ITEM 70: USE CHECKED EXCEPTIONS FOR RECOVERABLE CONDITIONS AND RUNTIME EXCEPTIONS FOR PROGRAMMING ERRORS

I follow the `Do Not Swallow The Exceptions` rule when I am faced with checked exceptions.

ITEM 71: AVOID UNNECESSARY USE OF CHECKED EXCEPTIONS

Didn't use any checked exceptions as they weren't needed.

ITEM 72: FAVOR THE USE OF STANDARD EXCEPTIONS

I create some of exceptions such as :

- `ConnectionOverflowException`, to represent that there are no available connections.
- `IllegalRequestException`, to represent that Illegal Request Type.
- `NotFoundRequestException`unhandled Request Type.

ITEM 73: THROW EXCEPTIONS APPROPRIATE TO THE ABSTRACTION

Not applicable to the project.

ITEM 75: INCLUDE FAILURE-CAPTURE INFORMATION IN DETAIL MESSAGES

The stack trace is always printed in the log in all the `try-catch` statements.

ITEM 76: STRIVE FOR FAILURE ATOMICITY

Not applicable to the project.

ITEM 77: DON’T IGNORE EXCEPTIONS

Exceptions are never ignored and well handled.

ITEM 78: SYNCHRONIZE ACCESS TO SHARED MUTABLE DATA

In classes `EmployeeCache`, `BasicConnectionPool` may be accessed and modified from multiple threads, so that i used blocking data structures.

ITEM 79: AVOID EXCESSIVE SYNCHRONIZATION

I used the minimum synchronized operations.

ITEM 80: PREFER EXECUTORS, TASKS, AND STREAMS TO THREADS

The multi-threading used to listen, send and receive data from / to other users ran on a ExecuterService with a dynamic number of threads to handle all users requests.

ITEM 81: PREFER CONCURRENCY UTILITIES TO WAIT AND NOTIFY

There was no need to use wait and notify or any other alternatives.

ITEM 83: USE LAZY INITIALIZATION JUDICIOUSLY

I used lazy initialization to initialize `DataManager`, `DBHelper` singleton objects, And to initialize `Repository`, `Gson` in local thread.

ITEM 84: DON’T DEPEND ON THE THREAD SCHEDULER

There was no depending on the thread scheduler.

Serialization

ITEM 85: PREFER ALTERNATIVES TO JAVA SERIALIZATION

I use Json Object to send / receive data between Client / Server.

ITEM 86: IMPLEMENT SERIALIZABLE WITH GREAT CAUTION

Not applicable to the project.

ITEM 87: CONSIDER USING A CUSTOM SERIALIZED FORM

Not applicable to the project.

ITEM 88: WRITE READOBJECT METHODS DEFENSIVELY

Not applicable to the project.

ITEM 89: FOR INSTANCE CONTROL, PREFER ENUM TYPES TO READRESOLVE

Not applicable to the project.

ITEM 90: CONSIDER SERIALIZATION PROXIES INSTEAD OF SERIALIZED INSTANCES

Not applicable to the project.